ANSWERS

A3.9 Project: mission control

Data structures · A level · OCR H446 1.4.2, AQA 7517 4.2.1.1, Eduqas A500QS 1.1 · about 30 min

What this lesson is about

A mission read from a file into a circular queue, moves looked up in a dictionary, position tracked as a vector, and an undo stack to bring the robot home.

- The brief
- The mission file
- Plan
- Step 1: the mission into a queue
- Step 2: moves as vectors
- Step 3: put it together

Questions

6 marks in all

1. A robot must be able to undo its moves, most recent first. Which data structure should hold the moves? [1 mark]

- ☐ **A** A stack
- ☐ **B** A queue
- ☐ **C** A hash table
- ☐ **D** A vector

> **Answer:** A. Undo needs last in, first out, which is exactly what a stack gives.

2. Commands arrive by radio faster than the robot can carry them out. They must be carried out in the order received, using a fixed amount of memory. Which structure fits best? [1 mark]

- ☐ **A** A circular queue
- ☐ **B** A stack
- ☐ **C** A dictionary
- ☐ **D** A linked list

> **Answer:** A. First in, first out in a fixed array whose slots are reused is a circular queue.

3. Which of these statements are true? [1 mark]

Tick every answer that is true.

- ☐ **A** A dictionary gives fast look-up of a value by its key
- ☐ **B** A stack returns items in the order they were added
- ☐ **C** Adding a displacement vector to a position vector translates the position
- ☐ **D** A hash table keeps its keys in sorted order

> **Answer:** A, C. A stack returns items in reverse order, and hashing scatters keys rather than sorting them.

4. This program works out a position by dead reckoning. What does it print? [1 mark]

```
DIRECTION = {"forward": [0, 1], "right": [1, 0], "backward": [0, -1]}
pos = [0, 0]
for word, cm in [("forward", 20), ("right", 15), ("backward", 5)]:
    d = DIRECTION[word]
    pos = [pos[0] + cm * d[0], pos[1] + cm * d[1]]
print(pos)
```

Answer:

```
[15, 15]
```

Each move adds cm times its unit vector: [0, 20], then [15, 0], then [0, -5], giving [15, 15].

5. Items go from a queue onto a stack and then off the stack. What does this program print? [1 mark]

```
queue = ["a", "b", "c", "d"]
stack = []
while queue:
    stack.append(queue.pop(0))
out = []
while stack:
    out.append(stack.pop())
print(out)
```

Answer:

```
['d', 'c', 'b', 'a']
```

The queue gives the items in order, and the stack gives them back last first, so the order is reversed.

6. Why does the mission program load every command into the queue before the robot moves? [1 mark]

- ☐ **A** A mission too long for the queue is found before anything happens
- ☐ **B** The robot cannot read a file while it is driving
- ☐ **C** A queue can only be filled once
- ☐ **D** It makes the queue sort the commands

Answer: A. If the queue fills while loading, the program can refuse the mission instead of stopping half way through it.

The task: mission control

Build the program from the brief. `slide(v)` is given: it drives the displacement vector `v = [x, y]` in cm (x to the right, y forward). Use fixed-size arrays with pointers for the queue and the stack, not the list's own `append`, `pop` or `insert`, and do not call `position()`. 1. Read every line of `mission.txt` (`<word>,<cm>`, where `<word>` is `forward`, `backward`, `left`, `right` or `abort`, and `<cm>` is a whole number) into a circular queue of size 8, wrapping the pointers with `%`, before the robot moves. 2. Keep a stack of size 8 with a `top` pointer, a dictionary `DIRECTION` from each direction word to its unit vector, and a position vector `pos` starting at `[0, 0]`. Do not compare the word with each direction name. 3. While the queue is not empty, dequeue a command: - A move: drive it with `slide`, push `(word, cm)`, add the move to `pos`, and print `at <pos>`, for example `at [0, 30]`. - `abort`: print `abort: <n> commands left in the queue`, where `n` is the number still in the queue. Then, until the stack is empty, pop a move, drive it in reverse, update `pos`, and print `undo <word> <cm>`, for example `undo forward 25`. Then stop taking commands. 4. Finally print `home at <pos>`.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def slide(v):
    """Drive by the displacement vector v = [x, y] in cm: x is sideways (right positive), y
is forward."""
    if v[0] > 0:
        right(50, distance=v[0])
    if v[0] < 0:
        left(50, distance=-v[0])
    if v[1] > 0:
        forward(50, distance=v[1])
    if v[1] < 0:
        backward(50, distance=-v[1])
```

The hint students can ask for: Load every line into the queue before you move. Then take commands off the front one at a time: a move is looked up in the dictionary, made, pushed onto the stack, and its direction vector, scaled by the distance, is added to the position. On abort, report the queue's count, then pop and reverse until the stack is empty.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

DIRECTION = {"forward": [0, 1], "backward": [0, -1], "left": [-1, 0], "right": [1, 0]}
OPPOSITE = {"forward": "backward", "backward": "forward", "left": "right", "right": "left"}

def slide(v):
    """Drive by the displacement vector v = [x, y] in cm: x is sideways (right positive), y
is forward."""
    if v[0] > 0:
        right(50, distance=v[0])
    if v[0] < 0:
        left(50, distance=-v[0])
    if v[1] > 0:
        forward(50, distance=v[1])
    if v[1] < 0:
        backward(50, distance=-v[1])
```

```
def add(a, b):
    return [a[0] + b[0], a[1] + b[1]]

def scale(k, a):
    return [k * a[0], k * a[1]]

SIZE = 8
queue = [None] * SIZE
front, rear, count = 0, -1, 0

def enqueue(item):
    global rear, count
    if count == SIZE:
        return False
    rear = (rear + 1) % SIZE
    queue[rear] = item
    count = count + 1
    return True

def dequeue():
    global front, count
    if count == 0:
        return None
    item = queue[front]
    front = (front + 1) % SIZE
    count = count - 1
    return item

stack = [None] * SIZE
top = -1

def push(item):
    global top
    if top == SIZE - 1:
        return False
    top = top + 1
    stack[top] = item
    return True

def pop():
    global top
    if top == -1:
        return None
    item = stack[top]
    top = top - 1
    return item

with open("mission.txt") as f:
    for line in f.read().splitlines():
        word, cm = line.split(",")
        enqueue((word, int(cm)))

pos = [0, 0]
while count > 0:
    word, cm = dequeue()
    if word == "abort":
        print("abort:", count, "commands left in the queue")
        while top != -1:
```

```
            word, cm = pop()
            step = scale(cm, DIRECTION[OPPOSITE[word]])
            slide(step)
            pos = add(pos, step)
            print("undo", word, cm)
        break
    step = scale(cm, DIRECTION[word])
    slide(step)
    push((word, cm))
    pos = add(pos, step)
    print("at", pos)
print("home at", pos)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.